A KNOWLEDGE-BASED SYSTEM FOR AUTOMATIC PROGRAM CONSTRUCTION

David Barstow
Computer Science Department
Stanford University
Stanford, California 94305

ABSTRACT: PECOS Is a system that synthesizes programs from abstract specifications through the *use* of a knowledge base of programming rules. The rules deal explicitly with various aspects of the programming process, including intermediate-level constructs and certain design decisions. Programs are specified as abstract algorithms in a high-level language for the domain of symbolic computation. Programs are constructed by a process of gradual refinement.- PECOS retrieves and applies relevant rules until the original abstract description has been refined into a concrete program. When several rules are relevant in the same situation, each is applied separately, enabling PECOS to construct a variety of programs for a given abstract specification.

KEYWORDS; Automatic programming, Program refinement, Programming knowledge, Rules about programming, Knowledge-based systems.

1. INTRODUCTION

PECOS is a system that uses a knowledge base of programming rules to synthesize programs from descriptions of abstract algorithms [Barstow 1977a]. The rules embody knowledge about a variety of techniques for implementing data structures and algorithms. This allows PECOS to construct several distinct implementations of the same abstract algorithm. The implementation techniques dealt with by the rules include linked list and array representations of sets, simple table-oriented techniques for representing correspondences, and several types of enumeration algorithms with varying degrees of efficiency.

Programs are specified in a high-level language whose constructs are drawn from the area of symbolic programming. These include such information structures as sets and correspondences and such operations as testing for membership in a set and computing the inverse of a correspondence. Programs are synthesized by a process of gradual refinement. PECOS repeatedly retrieves and applies rules from its knowledge base until the original abstract description has been refined into a concrete implementation in the target language. Currently the target language is LISP (in particular, a subset of INTERLISP [Teitelman 1975]). When several rules are relevant in the same situation, PECOS applies each separately, resulting In the synthesis of a distinct program for each rule. PECOS can thus construct several implementations from one specification.

This ability is central to the role of PECOS as the Coding Expert of the PSI () program synthesis system [Green 1976]. Loosely speaking, the Coding Expert's task is to enable the construction of a variety of implementations of the specifications produced by ^'s acquisition phase[1]. The other module of the synthesis phase, an Efficiency Expert known as LIBRA, is used to select from among these alternatives.

The development of PECOS has required research along four directions: (1) the codification of a body of knowledge about programming; (2) the development of a model of program construction appropriate for that knowledge; (3) the design of a representation scheme; and (4) the design and implementation of a system that can apply the knowledge to produce implementations of abstract algorithms. No one aspect can be totally isolated from any of the others, but this paper will focus on the last. Overviews of the knowledge base and the model of program construction will be given, followed by a brief description of the representation scheme. Finally, PECOS's control structure will be discussed and an example of PECOS in operation will be given.

2. THE KNOWLEDGE BASE

Most of PECOS's abilities are based on its access to a large store of rules about programming. A sample of these rules is presented below. For clarity, the rules are given in English; a brief discussion of the internal representation will be given later.

RULE1: A collection may be refined into an explicit collection.

RULE Pi A sequential collection may be refined into a linked list.

RULES: If a linked list is represented as a LISP list, a test of whether an item is stored in an element cell of the list may be refined into a call to the LISP function MEMBER.

RULE4-. If a collection is represented as a correspondence of items to boolean values, a test of whether an item is an element of the collection may be refined into a retrieval of the value corresponding to the item.

RULES: If there is an ordering relation for the elements of a sequential collection, a test of whether an item is stored in some location in the collection may be refined into an enumeration of the elements in the collection, in which the enumeration order is based on the relation and in which the enumeration may be terminated early if an item is found which follows the target item according to the relation.

RULE6: If the enumeration order is the same as the stored order of a collection, the state of the enumeration may be saved as a position in the collection.

RULE7: If the enumeration order is ordered by an ordering relation and the stored order in a sequential collection is ordered by the same relation, the enumeration order is the same as the stored order.

RULE8: If there is an ordering relation for the elements of a sequential collection, the elements may be stored according to that relation.

RULE9: An ordering relation for integers is "less than."

[1] PECOS's specification language is that used by ty's Program Model Builder [McCuna 1077].

RULE 10: A scheme for remembering a value computed in one place and used elsewhere is to bind the value to a variable.

RULE 11: If a value is remembered as the value of a variable whose name Is X, the value may be accessed by retrieving the value of X.

Several observations about the rules are In order. First, note that the rules mention various programming concepts explicitly. The most abstract concepts are those Included In PECOS's specification language (e.g., "collection", "correspondence", "Is-element"). The most concrete are those used In the target language (e.g., "LISP list", "MEMBER function"). Some concepts (e.g., "linked list", "position in a sequential collection") represent intermediate notions, more concrete than the specification language and more abstract than the target language. Other concepts (e.g., "enumeration order", "scheme for remembering a value") represent design decisions involved in symbolic programming. Among the benefits of such abstraction levels are significant amounts of knowledge-sharing (e.g., knowledge about the common aspects of linked lists and arrays Is dealt with at the level of sequential collections) and relatively simple extensibility (different abstractions provide a variety of hooks for new rules).

Second, note that refinement plays a central role in the rules. RULE2, for example, describes "linked list" as a refinement of "sequential collection." The essence of such a relationship Is that the refined concept (linked list) is a more concrete notion than the abstract concept (sequential collection), PECOS uses such rules to drive the refinement process which produces the final program: the application of RULE2 to a given sequential collection represents a decision to Implement it as a linked list. The validity of such an Implementation is often dependent on other parts of the program. For example, RULE4 specifies that a "get-correspondent" operation is a refinement of an "is-element" operation only if the collection has been refined into a correspondence.

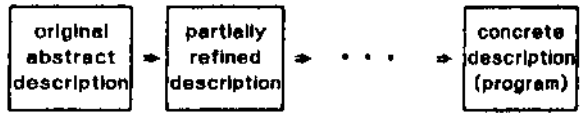
Third, note that the rules are all relatively small. One major motivation for designing the rules this way is that individual rules are intended to reflect individual decisions and choices, rather than groups of choices, thereby making it easier to focus on the nature of each decision. A second motivation Is that the rules are then relevant In wide ranges of situations.

Finally, note that some of the rules deal with programming in general and others *are* specific to programming in LISP. For example, RULE2 could be considered a general rule while RULE3 could be considered a LISP rule. One of the benefits of such a separation is that the replacement of the LISP rules by rules for some other language might enable PECOS to write programs In that language. A set of rules for SAIL (an ALGOL-Ilke language) Is currently being developed. A few simple SAIL programs have been written using a combination of previously developed general rules and the newly developed SAIL rules [Ludlow 1977].

Considerable effort has gone Into the codification of this knowledge base In the hope of developing a set of rules reflecting notions and decisions inherent in the programming process (although often In the guise of hidden assumptions). Taken together, the rules may be considered to be a step toward a "science" of programming: they are a coherent collection of facts, specifying certain concepts and relationships between them, which describe part of the process of programming within the domain of symbolic computation. The explication of such knowledge has been a major goal of this line of research [Green and Barstow 1976, 1975]. The PECOS system demonstrates that such a body of knowledge can provide the basis of an automatic programming system; in fact, the rules were developed with this application in mind. However, it is hoped that the concepts and relationships are sufficiently relevant to other aspects of programming that other applications may be found. An interesting possibility might be to investigate their usefulness in the teaching of programming. For the interested reader, a description of the entire collection of rules may be found elsewhere [Barstow 1977a].

3. A REFINEMENT MODEL OF PROGRAM SYNTHESIS

PECOS constructs programs through the application of rules from its knowledge base. This process Is based on a model of program synthesis as gradual refinement. This may be simply illustrated as a sequence of program descriptions:



Each description in the sequence is slightly more refined (concrete) than the previous description. The first is the program description In the specification language and the last is the implementation in the target language. Such sequences will be referred to as *refinement sequences* and the individual descriptions will be referred to as *program descriptions.* The process of deriving one description from the previous one will be referred to as a *refinement step.*

We may Illustrate this through a very simple example (more typical cases often require hundreds of refinement steps). Suppose the original specification is:

Is X (integer) an element
of Y (collection of integers)?

(In order to focus on the refinement process, we will use English phrases for program descriptions. The Internal form of these descriptions Is a collection of nodes and properties as will be discussed In more detail in the next section.) The first refinement step might be to decide to implement the collection (Y) as an explicit collection (as opposed to some form of implicit representation, such as upper and lower bounds):

Is X (integer) an element
of Y (explicit collection of integers)?

The next step might refine the "Is-element" operation. An appropriate refinement for explicit sets would be:

Is X (integer) explicitly an element
of Y (explicit collection of Integers)?

The next step might be to represent the explicit collection as a stored collection (as opposed to markings on the elements, for example):

Is X (Integer) explicitly an element
of Y (stored collection of integers)?

with a corresponding refinement for the operation:

Is X (Integer) stored as an element
of Y (stored collection of integers)?

The next decision might be to implement the stored collection as a sequential collection (in contrast with a tree representation):

> Is X (integer) stored as an element of Y (sequential collection of integers)?

Again, there is a parallel refinement for the operation:

> Is X (integer) stored in a location of Y (sequential collection of integers)?

Next, we might decide to implement the sequential collection as a linked list (rather than, say, an array):

> Is X (integer) stored in a location of Y (linked list of integers)?

with the corresponding refinement of the operation:

> Is X (integer) stored in an element cell of Y (linked list of integers)?

Note that up to this point, no use has been made of the fact that the target language is LISP. For example, the linked list could still be represented using a parallel array representation, one holding the items and one holding the links. However, in the interest of simplicity, we will let the next refinement step be the decision to use a LISP list implementation for the linked list:

> Is X (integer) stored in an element cell of Y (LISP list of integers)?

At this point, the refinement of the operation is dependent on the refinements of X and the elements of Y. We will suppose that both are implemented as LISP integers, producing the next two refinement steps:

> Is X (integer) stored in an element cell of Y (LISP list of LISP integers)?

> Is X (LISP integer) stored in an element cell of Y (LISP list of LISP integers)?

Finally, the operation can be refined into a call to the LISP function MEMBER:

> (MEMBER X Y)

We thus have a sequence of twelve refinements taking the original specification and producing a LISP expression that implements it. Note that the refinements at each step involve only very small changes. This is characteristic of the programming rules upon which PECOS is based. A deliberate effort has been made to identify and separate out decisions that are often implicit. Thus, the complete refinement of the collection Y took five steps, each corresponding to a particular design decision. Note also that some steps may require other steps to have been made previously. For example, the refinements which led from the "is-element" operation to a call to MEMBER each required the data structure to have been refined in a particular way. Although it is quite common for the refinements of operations and data structures to proceed in such a parallel fashion, it need not always be the case. One part of a program may be refined to great detail before another part is considered. In addition, several algorithms might be appropriate for the same data structure, just as the same algorithm may be applicable to several representations. For example, a linear scan with an explicit loop would have served just as well as the call to MEMBER.

Some of the rules shown in the previous section could be used to derive parts of the refinement chain we have just seen. For example, RULE1 could be used to refine the first diagram into the second, RULE3 could be used to produce the final refinement step, and so on.

4. INTERNAL REPRESENTATIONS

Before considering the details of how PECOS uses its knowledge base to produce refinement sequences, a few details of the internal representation must be presented.

4.1. Descriptions in a Refinement Sequence

Internally, PECOS represents each description as a collection of nodes. Each node is labelled with its CONCEPT. (As mentioned earlier, these concepts identify particular programming notions.) Thus, a node labelled IS-ELEMENT represents an operation testing whether some item is in a particular collection. In addition to the label, each of the nodes has a set of properties related to the node's concept. For example, an IS-ELEMENT node would have properties for each of its arguments. (The properties are named ELEMENT and COLLECTION.) Although property values may be arbitrary expressions, they are frequently links to other nodes (as in the IS-ELEMENT case). Figure 1 shows part of the internal representation for the expression 1S-ELEMENT(X,1NVERSE(Y,Z))[2].

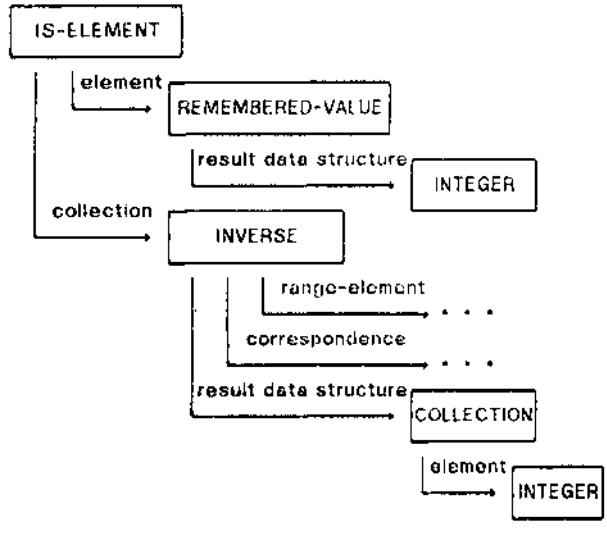


Figure 1

Fnch box is a node. The labels inside the boxes are the concepts and the labeled arrows are properties. In the refinement sequence presented earlier, the arguments to the IS-ELEMENT operation were given merely as X and Y. In general, such arguments are links to nodes representing operations which return values to be used as the arguments. (The values are indicated by the RESULT-DATA-STRUCTURE property.) In this case, one of

[2] Read "Is X an element of the inverse mapping of Y under the correspondence Z?"

these operations is a REMEMBERED-VALUE node, specifying that the value to be used is computed elsewhere in the program. The other operation is an INVERSE node. Note that the result of the inverse operation is represented explicitly by the COLLECTION node, although it is only implicit in the corresponding English description. This node represents the data structure passed from the inverse operation to the is-element operation. Rules for refining INVERSE generally have conditions on the data structure produced, to insure that the representation is what the refinement produces. Likewise, rules for refining IS-ELEMENT generally have conditions on the argument data structure in order to insure that the refinement can accept that representation for its arguments. Since the same node (the COLLECTION node) is used in both cases, the two refinements are coordinated.

There are only a few types of changes that may be made in a single refinement step: the introduction of new nodes, the addition of properties to existing nodes, and the refinement of one node into another[3]. Other types of changes (e.g., modification of an existing property) may be imagined, but no use for them has yet been found.

4.2. Rules in the Knowledge Base

The internal representation of the programming rules is based on two observations. First, the changes made in each refinement step are generally quite small and are of certain specific types. Second, the rules all have the form of condition-action pairs, where the conditions are patterns to be matched against subparts of descriptions, and the actions are instances of the types of modifications just mentioned. These observations lead to a classification of rules into a set of *rule types:*

Refinement rules refine one node pattern into another. The refined node is typically created at the time of rule application. These are the rules which carry out the bulk of the refinement process, and are by far the most common type. RULE1, RULE2. RULL3. and RULES are refinement rules.

Property rules attach a new property to an already existing node. RULE9 and RULE10 are property rules. Property rules are often used to indicate explicit decisions which guide the refinements of distinct but conceptually linked nodes. For example, RULE10 specifies how references to a data structure are to be refined.

Query rules *are* used to answer queries about a particular description. Such rules arc normally called as part of the process of determining the applicability *of* other rules. RULE7 is a query rule.

The internal representation of the rules is based on this categorization of rule types:

(REF- *<node pattern> <refincment node»*

(PROP- *<property name> <node pattern> <value»*

(QUERY- *<query pattern> <query answer»*

where REF-, PROP-, and QUERY- are tags indicating rule type. In REF- and PROP- rules, each *<node pattern>* consists of a *<conccpt>* and an *<applicability pattern>.*

[3] Although it is not strictly accurate, "refinement" may bo considered to mean the replacement of the more abstract node by the refined node. For various reasons, PECOS actually retains the abstract node as well as the refinement node.

Several other issues have arisen in the design of the rule base organisation, including an indexing scheme for efficient rule retrieval, the details of the pattern matcher, and the breakdown of the condition patterns into separate parts for different uses. Detailed discussions of these issues may bo found elsewhere [Barstow 1977a, 1977b].

It should be noted that several kinds of rules are not easily expressed in the current formalism. For example, more general inferential rules (such as the test constructor used by Manna and Waldinger [Manna *and* Waldinger 1077]) and rules about certain kinds of data flow (such as merging an enumeration over a set with the enumeration that constructed it) can not be described conveniently with the current set of rule types. It is not clear how difficult it would be to extend PECOS to handle such cases.

5. CONTROL STRUCTURE

PECOS uses a relatively simple task-directed control structure to develop refinement sequences for a given specification: in each cycle, a task is selected and a rule applied to the task. When several rules *are* applicable, a refinement sequence can be split, with each rule applied in a different branch. PECOS thus actually creates a tree of descriptions in which the root node is the original specification, each leaf is a program in the target language, and each path from the root to a leaf is a refinement sequence*. While working on a given task, subtasks may be generated; these are added to the agenda and considered before the original task is reconsidered.

There are three types of tasks (corresponding to the types of changes involved in refinement steps and to the different rule types):

(REFINE *n)* specifies that node *n* is to be refined.

(PROPERTY *p n)* specifies that property *p* of node *n* is to be determined.

(QUERY *rel argl argZ* ...) specifies that the query *(rel argl arg2* ...) must be answered.

When working on a task, relevant rules are retrieved and tested for applicability. For example, if the task is (REFINE 72) and node 72 is an IS-ELEMENT node, then all rules of the form (REF-(IS-ELEMENT...).. .) will be considered. When testing applicability, it may be necessary to perform a subtask. For example, an argument may need to be refined in order to determine if it is represented as a linked list. This is, in fact, quite common: the refinement of one node is often the critical factor making several rules inapplicable to a task involving another node.

With its current knowledge base, PECOS usually succeeds at achieving each task. (Hence, most refinement sequences lead to complete programs.) The major reason for this lies in the knowledge base: there are refinement rules for each intermediate level concept, and property rules for each of the necessary properties. However, some sequences lead to situations where no rules are applicable, and PECOS abandons such linos of development.

In the interests of brevity, further details of the control structure and context mechanism used for the tree of descriptions will not be considered here. More detailed discussions may be found elsewhere [Barstow 1077a, 1977b].

[fl] The nature of such refinement trocs is central to the practicality of this approach and will be discussod In section 7.

6. PECOS IN OPERATION

Lot us now look at PECOS implementing a different algorithm for the 1S-ELEMENT operation, in order to illustrate the nature of the tasks and the way that PECOS focuses design decisions. Recoil that the original specification was;

Is X (integer) an element
of Y (collection of integers)?

We will pick up the refinement process at the point where Y has been refined into a SEQUENTIAL-COLLECTION, and where the 1S-ELEMENT operation has been refined into an IS-STORED-IN-A-LOCATION operation. The node and pioperty representation is given in Figure 2 (with several nodes omitted in order to emphasize the important ones). The numbers on the left will be used to refer to the nodes.

The task is now (REFINE 1), that is, to refine the IS-STORED-IN-A-LOCATION node. In the example in section 3, we followed the path where this was refined into a test on whether X was stored in an element cell of Y. Another refinement rule, RULE5, is applicable here, but only if there is an ordering relation for the elements of the sequential collection Y, so a subtask is established: (PROPERTY ORDERING-RELATION 4). One rule for this task, RULF9, is applicable only if the items are integers, as is the case here. PECOS applies this rule and LESS-THAN is attached as the ORDERING-RELATION property of node 4. Having achieved the subtask, PL"COS reconsiders the original task (REFINE 1), and RULES is now applicable. This produces an ENUMERATE-ITEMS node, where the enumeration is constrained to be in the order specified by the ordering relation associated with the elements of Y (LESS-THAN) and where the search may be terminated prematurely (before all the elements of Y have been examined) if an element is found which follows X under the same ordering relation. Constraints sue!) as these are specified by attaching properties to the ENUMERATE-ITEMS node, as illustrated in Figure 3.

In turn, the ENUMERATE-ITEMS node (5) is refined into a loop structure which uses an ENUMERATION-STATE node to determine how far the enumeration has proceeded. The refinement of this node coordinates the refinements of the parts of the program that refer to it (initialization, termination test, incrementation), as illustrated in Figure 4. Note how nodes 7, 8, and 9 all have links to node 6. This is typical of the way that PECOS focuses design decisions: the refinements of the various parts are all dependent on a single node.

When PECOS tries to refine the ENUMERATION-STATE node, RULE6 is applicable only if the elements are to be enumerated in the order in which they are stored. This causes a subtask to be generated: (QUERY STORED-ENUMERATION-ORDER 6). The ENUMERATION-ORDER has already been constrained to be ORDERED by the LESS-THAN relationship. All that is needed, then, is to determine whether the sequential collection is also ordered. The subtask for this is: (PROPERTY STORED-ORDER 3). Two rules are applicable here, and we will follow RULE8, which specifies that the items may be ordered if there is an ordering relation for them. Since an ordering relation, LESS-THAN, has already been found, RULE8 can be applied, resulting in (ORDERED LESS-THAN) as the STORED-ORDER property of node 3, a

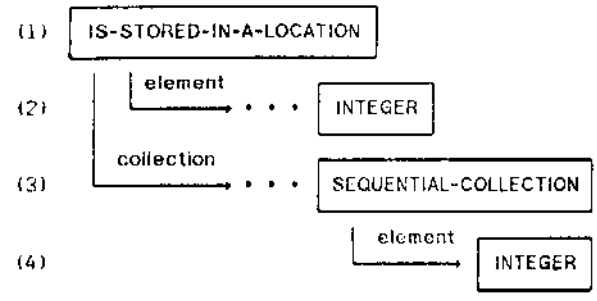


Figure 2

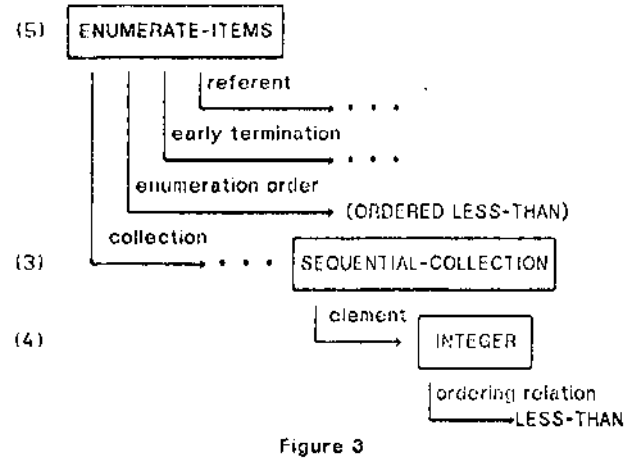


Figure 3

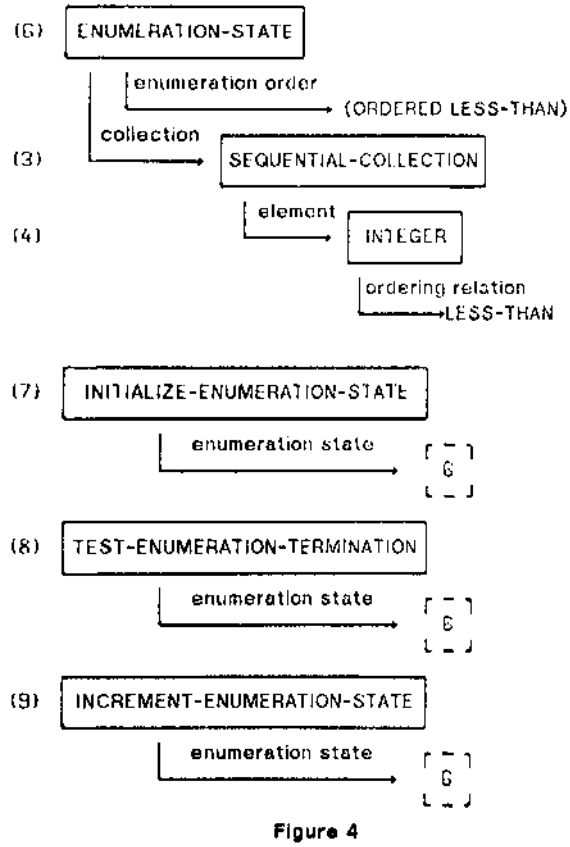


Figure 4

further specification on the representation of Y[6]. Now RULE7 can provide the answer, TRUE, to the query, and RULE 6produces a P0SIT10N-1\[T]-C0LLECTJ0N node as a refinement of node 6. The rest of the refinement path is relatively straightforward. Note, however, that at this point Y still has not been refined further than a sequential collection. AJl of the rules that have been used are independent of the particular implementation chosen for the sequential collection; linked lists and arrays would work equally well. This is an example of the v/ay the rules in the knowledge base have been explicitly designed to operate at the highest possible level of abstraction. Before the program can be completed, of course, this decision must be made, and we will assume that, again, a linked list is chosen. This is the last interesting aspect of the refinement sequence. For the curious, the final program looks like the following (with mnemonic labels for increased readability):

```
(PROG (REST ITEM)
      (SETQ RLST Y)
  RPT (SETQ ITLM (CAR REST))
      (COND
        ((OR (NULL REST)
             (ILESSP X (CAR ITEM)))
          (RETURN NIL))
        ((EQP (CAR ITEM) X)
          (RETURN T)))
      (SETQ REST (CDR REST))
      (GO RPT))
```

If the IS-ELEMENT operation were embedded in the specification of a Jarger program, the decision to represent Y as an ordered sequential collection would presumably have wider ramifications, possibly including the necessity of sorting Y. This Is characteristic of the programs that PECOS writes: multiple representations are often considered for what In the original specification was the same data structure, with the program converting representations as necessary.

7. A SPACE OF PARTIALLY IMPLEMENTED PROGRAMS

One of the motivations for this work is the observation that different implementations of abstract concepts are appropriate in different situations. PECOS's refinement tree represents a space that can be searched for such an "appropriate" implementation. In fact, the ability to construct such a space was one of the primary goals in developing PECOS's rules[6]. Since determining the appropriate implementation involves exploring this space, the sizo of PECOS's refinement trees is a critical issue.

One aspect of the size is the length of the individual refinement sequences (i.e., the number of rule applications). This varies greatly, but preliminary experiments indicate that about two rule applications are required per CONS cell in the S-expression for the final program. Thus, if the correct choice can be made without expanding any alternative paths, the cost of constructing the best implementation is roughly linear with the size of the final program. This cost may be reduced through the use of special purpose rules to handle frequently occurring cases. Such rules can collapse an entire sequence of rule applications into a single rule. While the rules to derive the sequence must still be available for use in other situations, one can frequently get by with a "short-cut" rule. PECOS currently has a few such rules and more will be added soon.

In an effort to reduce the entire space, PECOS currently postpones consideration of all choicepoints for as long as possible[7]. This technique, together with the use of several levels of abstraction, results in trees that tend to be "skinny" at the top and •'bushy" at the bottom. In the membership test, for example, by delaying the refinement of the sequential collection (into either a linked list or an array), 1 ?6 refinement steps were made before working on the choice; following the split, 16 further steps were required in the linked list case and 51 steps in the array case. Had the split been made when the task was first considered, splitting would have occurred after 9 steps, leaving 133 left in the linked list case and 168 in the array case. Thus, delaying the choicepoint cut the total number of rule applications in half.

The techniques mentioned above are intended to reduce the total size of the space. Another simple technique helps to reduce the amount of the space that must be searched. When two parts of the program arc totally independent, and both involve choices, the entire space represents the cross-product of the alternatives in, the two cases. However, under the assumption that the independence of the parts enables a choice for one to be made independently of any work that has been done on the other, the exploration of some of the paths can be avoided. The basic idea is that one of the choicepoints be carried through to conclusion and one of its alternatives be chosen before considering the other choicepoint. Then the alternatives for the second choicepoint need only be considered in the chosen path for the first.

Although these simple techniques can help to reduce the size of the space, the problem of actually making choices still remains. Perhaps the greatest benefit of the *use* of abstraction and refinement is the way that such choice-making is facilitated. Since each step is quite small, the effects of alternative branches can be relatively easily understood. It is usually unnecessary to complete all implementations in order to determine which alternative is preferable. In addition, the uniform representation of descriptions in refinement sequences facilitates the use of analytic planning techniques based on cost estimators for the unrefined parts. Work on LIBRA Ol' 's Efficiency Expert) has included the development of such heuristic and analytic techniques for making choices [Kant 1977]. PECOS and LIBRA have been interfaced and now operate as ty's synthesis phase. A discussion of their interaction, as well as a detailed example of their joint operation, is available elsewhere [Barstow and Kant 1076],

Several other artificial intelligence programs have employed similar tactics (e.g., NOAH [Sacerdoti 1075]).

[5] Note that when that the decision is taken the other way (to leave the elements unordered), the refinement path requires the construction of a considerably more complicated enumeration, one which searches the collection at each stage for the smallest element, then compares that to X, and so on. The result is an implementation of the IS-ELEMENT operation that requires on the order of n^2 comparisons!

[6] Note that the nodes of this space all represent somantically correct programs; that is, they all implement the original specification, although at varying levels of detail. The use of refinement rules enables PECOS to proceed directly toward correct implementations, without any need for testing different combinations of primitive constructs to see if they are correct.

8. DISCUSSION

The use of large amounts of detailed knowledge has become on increasingly important paradigm for developing artificial intelligence systems. However, most approaches to automatic programming have concentrated on the development of relatively general techniques (e.g., SYNSYS [Manna and Waldinger 1977]). PECOS'S rules enable the construction of larger and more varied programs, but the price paid is a loss of generality. The "ultimate" automatic programming system will presumably employ both techniques. Although current trends in programming methodology emphasize abstraction and refinement, most automatic programming work has not used intermediate level concepts extensively. Low's data structure selection system involves mapping directly from abstract concepts to machine-level constructs [Low 1974], Most problem-solvers deal directly with the primitive operations of the target language.

Currently (May 1977), there are about two hundred rules in PE COS's knowledge base. In the next few months, this number will increase to about three hundred. This enlargement of the rule base will concentrate on higher level knowledge (e.g., alternative set representations), as the current rules involving low level knowledge should be able to handle most of the low level details. Even with its current rule base, PECOS is able to construct a variety of implementations for specifications of many different programs. In theory, PECOS could construct implementations for any program expressible in the high-level specification language. In practice, time and space limitations prevent the consideration of programs whose specification is more than about "a page." The number of implementations possible for a given program specification varies considerably with many factors (for instance, whether a particular primitive is known to be numeric or symbolic), so the best indication is perhaps to take a simple example. PECOS can implement the "is-clcmcnt" program considered in this paper in about a dozen ways, with the principle variations being the use of linked lists, arrays, or very simple hash tables for the data structures, and the use of several types of enumeration for the algorithms. The most complex program that PECOS has implemented is one involving manipulation of an inverted correspondence (i.e., indexed by range values). The specification was about a half page in length and the implementation was about 100 lines of INTERLISP code.

The development of PECOS represents the final stage in an experiment investigating a knowledge-based approach to automatic program construction. The essence of this approach , involves the identification of concepts and decisions involved in the programming process and their codification into individual rules. These rules are then represented in a form suitable for use by an automatic programming system. The benefits of this approach seem to lie in the variety of programs (with respect to both data structures and algorithms) that can be implemented. A side benefit is that the rules themselves constitute a detailed explication of knowledge about symbolic programming. PECOS demonstrates that such a knowledge base can *serve* as the basis of an automatic programming system capable of constructing a variety of implementations for abstract specifications.

9. ACKNOWLEDGEMENTS

Cordcll Green, my thesis advisor, and the other members of the ty project have been a source of motivation and focus for this work. Interaction with Elaine Kant and her work on LIBRA has been especially beneficial. Juan Ludlow's development of rules for SAIL exposed some of the hidden assumptions in my rules. Brian McCune contributed greatly to the design of the specification language. Randy Davis and Jorge Phillips have provided very helpful comments on earlier drafts of this paper.

10, REFERENCES

[Barstow 1977a]
Barstow, David R. *Automatic construction of algorithms and data structures using a Knowledge base of programming rules,* forthcoming Ph.D. thesis, Stanford University, 1977.

[Barstow 1977b]
Barstow, David R. *A knowledge base organization for rules about programming.* Workshop on Pattern-directed Inference Systems, May 107/ (to appear in SIGART, June 1977).

[Barstow and Kant 1976]
Barstow, David R., and Kant, Elaine. *Observations on the interaction of coding and efficiency knowledge in the PSI program synthesis system.* Proceedings of the Second International Conference on Software Engineering, San Erancisco, California, October 1976, pages 19-31.

[Green 1976]
Green, Cordcll. *The design of the PSI program synthesis system.* Proceedings of the Second International Conference on Software Engineering, San Francisco, California, October 1976, pages 4-18.

[Green and Barstow 1976]
Green, C. Cordell, and Barstow, David R. *A hypothetical dialogue exhibiting a knowledge base for a program understanding system,* in Elcock, E. W., and Michie, D. (Eds.). Machine Representations of Knowledge. Ellis Horwood Ltd. and John Wylie, 1976.

[Green and Barstow 1975]
Green. C. Cordell, and Barstow, David R. *Some rules for the automatic synthesis of programs.* Advance Papers of the Fourth International Joint Conference on Artificial Intelligence. Tbilisi, Georgia, USSR, September 1975, pages *232-239.*

[Kant 1977]
Kant, Flaine. *7he selection of efficient implementations for a high lei/el language.* Proceedings of ACM SIGART-SIGPLAN Symposium on Artificial Intelligence and Programming Languages, August 1977.

[Low 1974]
Low, J. *Automatic coding: choice of data structures.* Stanford University, Computer Science Department, AIM-242. August 1974.

[Ludlow 1977]
Ludlow, Juan. Masters Project. Stanford University, 1977.

[Manna and Waldinger 1977]
Manna, Zohar and Waldinger, Richard. *The automatic sytithcsis of recursive programs.* Proceedings of ACM SIGART-SIGPLAN Symposium on Artificial Intelligence and Programming Languages, August 1977.

[McCune 1977]
McCune, Brian P. *The PSI program model builder: synthesis of very high-level programs.* Proceedings of ACM SIGART-SIGPLAN Symposium on Artificial Intelligence and Programming Languages, August 1977.

[Sacerdoti 1975]
Sacerdoti Earl. *The nonlinear nature of plans.* Advance Papers of the Fourth International Joint Conference on Artificial Intelligence. Tbilisi, Georgia, USSR, September 1975, pages 206-214.

[Teitelman 1975]
Tcitelman, Warren. INTERLISP Reference Manual. Xerox Palo Alto Research Center, Palo Alto, California, December 1975.

INFORMALITY IN PROGRAM SPECIFICATIONS

Robert Balzer, Neil Goldman and David Wile
Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina Del Rey, California 90291

ABSTRACT

This paper is concerned primarily with (1) the procedure by which process-oriented specifications are obtained from goal-oriented requirement specifications and (2) computer-based tools for their construction. It first determines some attributes of a suitable process-oriented specification language, then examines the reasons why specifications would still be difficult to write in such a language. The key to overcoming these difficulties seems to be the careful introduction of informality based on partial, rather than complete, descriptions and the use of a computer-based tool that uses context extensively to complete these descriptions during the process of constructing a well-formed specification. Some results obtained by a running prototype of such a computer-based tool on a few informal example specifications are presented and, finally, some of the techniques used by this phototype system are discussed.

/. *INTRODUCTION*

A critical step in the development of a software system occurs when its goal-oriented requirements specification is transformed into a process-oriented form that specifies how the requirements are to be achieved. Only after this transformation has occurred can the feasibility of the system be analyzed and the consistency of the process specification with the requirements be verified. The key to this transformation is expressing the process-oriented specification abstractly so that its functionality is completely determined while the class of possible implementations remains broad.

We believe that such abstract process-oriented specifications are the key to rationalizing the software development process. Such specifications are, in reality, programs written in a very high level abstract programming language. As such, they could provide an effective interface between the two major software concerns: functionality and efficiency. These concerns should be decoupled so that the functionality of a system can be addressed before its efficiency has been considered. Once functionality has been accepted, it can be preserved while the system is optimized. Thus, since the abstract process-oriented specification is a program, its consistency with the requirements could be formally verified, informally argued, or tested by actually executing the specification. Furthermore, the end user could be given hands-on experience exercising the specification to see if it behaved as intended. Deviations and/or inconsistencies could be corrected in the specification before any implementation occurred.

Once the system's functionality has been accepted by the user, the efficiency of the system in meeting its performance requirements remains an issue. Such efficiency must be gained without altering the system's accepted functionality. We have argued elsewhere [1] that a computer-based tool can be built which guarantees maintenance of functionality while a program is optimized without sacrificing the programmer's ingenuity or initiative in determining how best to achieve efficiency.

NOTE: This research was supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. DAHC15 72 C 0308, ARPA Order No. 2223, Program Codes 3D30 and 3P10.

In this paper we are concerned primarily with the procedure by which such process-oriented specifications are obtained and with computer-based tools for their construction. We will begin by determining some attributes of a suitable process-oriented specification language, then examine why specifications would still be difficult to write in such a language. We will argue that the key to overcoming these difficulties is the careful introduction of informality based on partial, rather than complete, descriptions and the use of a computer-based tool which utilizes context extensively to complete these descriptions during the process of constructing a well formed specification. We will then present some results obtained by a prototype of such a computer-based tool on a few informal example specifications. Finally, we will discuss some of the techniques used by this prototype system.

2. ATTRIBUTES OF SUITABLE PROCESS-ORIENTED SPECIFICATION LANGUAGES

As stated above, a suitable process-oriented specification must completely define functionality, represent a broad class of possible implementations, and be executable.

How can we obtain such a language? We begin by noting that a suitably abstract programming language is a specification language. Several recent languages almost meet the above requirements for an executable specification language. They have arisen from two separate disciplines:

1. *Specification Languages.* Languages, such as RSL[2], PSL[3], etc., designed specifically for specification, describe a system in terms of data flows *and* processing units but do not functionally define the processing. Such languages can provide a simulation of the described system down to some level of detail, but cannot describe or simulate its full functionality.

2. *Abstract programming Languages.* Spawned by Dijkstra's notions of structuring, a generation of programming languages (CLU[4], Alphard[5], Euclid[6], Pearl[7]) has bloomed which isolate the definition of data objects, and the operations allowed on them, from their use and manipulation in the program. The result is the ability to use abstract program entities which model those that occur in the application being programmed. These entities are defined in terms of more computer-science-oriented entities, which are, in turn, defined in terms of more primitive ones, until the primitive objects and operations of the language are reached. Without the successive refinements of the abstract objects and operations, these languages would be suitable for specification, except that they would then lose their property of executability. Their executability has been gained at the expense of complete specification of implementation (down to the base level of the language).

What is clearly needed, then, is a language which can fully specify a system functionally without fully specifying its implementation. What are the required properties of such a language?

First, it must be able to define and manipulate application-oriented objects (as is done by the abstract programming languages). Second, the description of these objects and operations must be in terms of some formalism that does not require successive refinement to gain functionality and that does not overly constrain the

implementation. This is the Key issue that would enable specification and programming languages to be unified.

Three formalisms have been proposed for this role: sets, axiomatic specification, and relational data bases.

One of the earliest efforts is Jack Schwartz's SETL[8] language. Sets are the single abstract type allowed for which multiple implementations exist. All the operations on sets can deal with any of the implementations. Thus, users need not be concerned with any of these implementations while specifying the manipulations to be performed on their sets. Because functionality was completely captured by the SETL definitions of sets, implementation did not have to be considered. However, such implementation-free functionality existed only for sets and was not extensible.

More recently, Guttag, Horowitz, and Musser [9] have discussed an axiomatic specification technique in which the functional behavior of new abstract objects are axiomatically defined by algebraic equations. These algebraic equations act as functional requirements which any implementation of the objects and operations upon them must satisfy. Furthermore, they provide a way of executing programs using the operations directly without providing any implementation. Whenever *an* operation is performed on an object, the "state" of that object is transformed by applying the algebraic equation for that operation to the existing "state." The resulting state is just another expression in the algebra. As more and more operations are performed, these states become more complex. However, the states can be simplified by general rules of the algebra such as AND(A False)->False, or by using the equations for the abstract objects as rewrite rules, such as for a stack, POP(PUSH(A x))=>A. Such equivalence rules are part of the functional definition of the operations on the abstract objects. If the axiomatic functional definitions are complete, then specifications in this language can be directly executed while no implementation need be selected and the choice of possibilities has not been constrained. These axiomatic functional definitions provide a user the capability of adding arbitrary new abstract types to the language that can be manipulated in an implementation independent way. This extensible capability is exactly analogous to SETL's built-in capability to manipulate sets in an implementation-independent way.

Finally, we have languages in which the "state" is represented by a series of assertions in a relational data base, rather than by an expression, and in which the effects of an action are expressed as a series of additions or deletions to the data base rather than as an equation to be applied to the "state." The big difference between these two approaches is that in the axiomatic approach the functional definitions are expressed as interactions between the operations on a data type and hence do not rely on any more primitive notions. In the relational approach, as in SETL, each operation is functionally defined in terms of how it affects a built-in primitive notion, the relational data base.

The self-defining, or closed, property of axiomatic definitions would seem to favor that approach because each abstract object and its operations can be considered in isolation without relying on outside semantics and without specifying any constraints on the implementation. Unfortunately, this property comes at the expense of expressing the behavior of objects entirely in terms of the operations upon them and the need to express this behavior in the form of algebraic equations so that the equivalence of alternative sequences of operations can be formed (e.g., the POP(PUSH(A x))->A equivalence cited earlier for stacks).

In the relational approach, rather than stressing a closely knit set of types and operations on them, objects are perceived entirely in terms of their relationships with each other and a set of primitive operations which allow these relationships to be built and destroyed and to be extracted. Non-primitive operations exist on the objects, but they merely alter the set of relationships that exist between the objects. This view allows incremental elaboration of objects, their relationships with each other, and operations upon them. Most importantly, this approach enables objects and operations to be modeled almost exactly as they are conceived by the user in his application (as measured by how they are expressed in our most unconstrained form of communication-natural language).

This latter property is the reason we have selected the relational approach: We feel it minimizes the difficulty that a user would have in constructing an operational specification.

3. WHY OPERATIONAL SPECIFICATIONS ARE HARD TO CONSTRUCT

Unfortunately, even when the user's difficulties in constructing operational specifications are minimized by the use of the relational approach, the task remains burdensome and error-prone, primarily because although a suitable language has been chosen, it is still formal. Each reference to an object or action must be consistent and complete. The large number of interacting objects, actions, and relationships require the user to do a great deal of (error-prone) clerical bookkeeping which impedes his attention to the specification itself and reduces its reliability.

Suppose we constructed a computer aid which relieved the user of these clerical chores. How would the specification task be altered? We begin by considering how people specify software systems when unconstrained by computer formalisms.

4. SEMANTIC CONSTRUCTS IN NATURAL LANGUAGE SPECIFICATION

We studied many actual natural language software specifications. The main semantic difference between these specifications and their formal equivalent is that partial descriptions instead of complete descriptions are used. When such partial descriptions are understood it is because they can be completed from the surrounding context. The partial descriptions focus both the writer's and the reader's attention on the relevant issues and condense the specification. Furthermore, the extensive use of context almost totally eliminates bookkeeping operations from the natural language specification. These are some of the properties we find so useful in natural language specifications and which we so sorely lack in formal specification languages.

We have evidence [see sections 5 and 6], in the form of a running prototype system that these properties can be added to a previously formal specification language and that a computer tool can complete the partial description from the existing context. Such a capability is not totally new; it already exists in limited form.

Most programming languages use the context provided by declarations to complete partial descriptions of the operations to be performed on those objects (e.g., ADD becomes either INTEGER-ADD or FLOATING-ADD, depending on the declared attributes of its operands). The Codsyl DBDTG report [10] goes further in the use of context by completing partial references to an item by use of the "current" instance of that item as established by some other statement in the program. Data base declarations are also used to determine how various

program variables *are* to be used in completing partial descriptions of data base items.

These uses of context in programming languages have been accepted, and even championed, because for each use, the context-providing mechanisms are well-defined, the completion rules are simple and direct, and only a single interpretation is valid.

The context mechanisms we are proposing here are much more complex, the context generated much more diffuse, and a given partial description may produce zero, one, or several valid interpretations. Zero valid interpretations means that the partial description is inconsistent with the existing context. A single valid interpretation means that the partial description can be unambiguously completed through use of the existing context. Multiple valid interpretations indicate that sufficient context does not exist to complete the description and that interaction with the user is required to resolve the ambiguity.

Our work should be viewed as an effort to provide more general context mechanisms to resolve the ambiguity introduced in the specification by partial descriptions. If, as we believe, such mechanisms can be provided, would they be a beneficial addition to specification languages?

5. *DESIRABILITY OF INFORMALITY*

We recognize that our approach is controversial and apparently opposes the current trend to make program specifications more and more formal and to introduce such formalisms earlier in the development cycle. We believe closer examination will reveal that our approach is not only compatible with the desire for increased formalism, but a necessary adjunct to it.

Attention has been focused on formalisms for program specification to the exclusion of concern with the difficulty and reliability of creating such formal specifications and with maintaining them during the program life-cycle. Our approach specifically addresses these issues.

First, it should be recognized that informality will always exist during the formulation of a specification. The issue is whether the informal form is explicitly entered into the computer and transformed, with the user's help, into the formal specification, or whether it exists only outside the computer system in someone's head or written somewhere in unanalyzable form. We should consider, then, the feasibility and the desirability of a computer-based tool to aid in the transformation of an informal specification into a formal one.

Let us begin with the question of feasibility. While the results presented in the next section are preliminary and the examples chosen far smaller and simpler than real specifications, we are optimistic about continued progress and ultimate practicality of this approach. However, since these results are far from conclusive, we invite the reader to reach his own conclusions after considering the examples of the next section and the description of the prototype system which follows them.

Assuming for the moment that such a system is feasible, we consider its desirability. Informal specifications have three obvious advantages. First, they are more concise than formal specifications and focus both the specifier's and the reader's attention. They are more concise because only part of the specification is explicit; the rest is implicit and must be extracted from context. Attention is focused on the explicit information and, therefore, away from the implicit information, which increases both the readability and the understandability of the specification.

The second advantage is that informal specifications which employ partial rather than complete descriptions are a familiar, in fact normal, mode of communication. This reduces the training requirements of users, permits a wider set of users, and reduces dependence on the judgment and accuracy of intermediaries.

The final advantage deals with the maintainability of the system. Since about 707. of the total life cycle costs of large systems are for maintenance, any improved capabilities in this *area are very* significant. As we have argued elsewhere [1], the main deterrent to maintainability is optimization. Optimization spreads information throughout a program and increases its complexity through increased interactions among the parts. Both of these optimization effects greatly impede the ability to alter the program. An obvious solution is to alter an unoptimized specification and then reoptimize the program. No cost-effective and reliable technology currently exists for such reoptimization, though one has been proposed

en

A similar situation exists between the informal and formal specifications. The creation of a formal specification involves spreading implicitly specified information throughout the specification and increasing the complexity by structuring the specification into parts and establishing the necessary interfaces between them. As before, both of these formalization effects greatly impede the ability to modify the specification. Again, a solution is obvious: modify the informal specification and retransform it into a revised formal specification. Under the assumed feasibility of our approach, this solution would be possible and would greatly simplify maintaining the formal specification of the system.

We now consider three possible disadvantages of a computer-based tool to aid in transforming an informal specification into a formal one. The first possible disadvantage is that the informal constructs will be misunderstood by the computer tool. This is entirely possible, just as it is when a human intermediary interprets an informal specification. While the computer tool cannot match human performance in understanding the informal specification, it operates much more methodically. It can question the user when it detects that there *are* alternative interpretations of some statement. It can record and make explicit all assumptions it makes in transforming the formal specification. It can paraphrase the informal specification to verify that its interpretation is accurate (the current prototype system records its assumptions and interacts with the user to determine the correct interpretation of unresolved ambiguities, but does not yet contain any paraphrase capabilities). Thus, feedback and interaction with the user can eliminate the problem of possible misinterpretation of the informal specification.

The second possible disadvantage is that the computer-based tool will decrease the reliability of the transformation to a formal specification. If the informal specification exists only outside the computer system, then we must rely on the accuracy of the user or, more often, on some trained intermediary to accurately transform it into a formal specification. This transformation depends upon properly understanding the informal specification (see previous paragraph), then restating it in the required formalism. Once the proper understanding has been obtained, the restatement involves moving information from one place to another and changing its form. History would indicate that such clerical bookkeeping transformations are error-prone and can always be done more reliably by a computer tool. Hence, once the

correct interpretation has been obtained through the use of context and interaction with the user, the restatement of the informal specification into the required formalism can be more reliably performed by the computer-based tool than by the user or his intermediary. Therefore, reliability would be improved rather than reduced by such a tool once understanding was obtained.

Understanding, rather than reliability, thus emerges as the key feasibility issue. One way to improve understanding is to increase the interaction with the user. This leads to the third possible disadvantage: that the required volume of interaction will abrogate the advantage of informality. We do not expect this to be an issue with the current system or its successors, since we feel that its current performance level, as evidenced in the following section, indicates that the required interaction rate would be sufficiently small to prevent annoying or sidetracking the user.

Thus, we conclude that the availability of such a computer-based tool would be highly desirable because it would simplify the creation of a formal specification while increasing the reliability of the formulation process; improve the maintainability of the formal specification; reduce special training requirements; and expand the base of potential users. The question of feasibility, which remains as the paramount issue, rests clearly on the ability to correctly interpret an informal specification. We therefore now present some preliminary results obtained by the prototype system and describe its operation so that the reader can observe its performance level and judge for himself the generality of its context resolution mechanisms and therefore its feasibility.

6. *RESULTS*

This section presents two examples successfully handled by the prototype system. The examples were extracted from actual natural language specification manuals, and the results illustrate the power of the system's context mechanisms. However, our system is a prototype and, as such, it is far from complete. New examples currently expose new problems which are resolved by adding new capabilities to the system. Therefore, until some measure of closure is obtained, it should not be assumed that the prototype will correctly process new examples of the same "complexity" as earlier examples. Our goal is to add each new capability in as general a form as possible so that when it is used in new examples it will function correctly. In this way we expect to "grow" the system as more complex and varied examples are tried.

For each of the examples, we present three figures: the actual parenthesized version of the informal input currently used by the system (to avoid syntactic parsing problems)[II], a manually marked version which indicates some of the informalities to be resolved by the system, and a stylized version of the formal output program produced by the system.

The first example is a system which automatically distributes messages to offices on the basis of a keyword search of the text of the message. Figure 1 gives the informal natural language description. Figure 2 indicates some of the imprecisions contained in this example which must be resolved to obtain the system's formalization of this specification as an operational program (Figure 3).

To give some measure of the amount of imprecision in this example and, therefore, the amount of aid provided by the system, we have compiled the following statistics:

Number of missing operands	■	18
Number of incomplete references	■	22
Number of implicit type conversions	•	9
Number of terminology changes	•	3
Number of refinements or elaborations	-	2
Number of implicit sequencing decisions	-	7

ACTUAL INPUT FOR MESSAGE PROCESSING EXAMPLE

♦((MESSAGES ((RECEIVED) FROM (THE "AUTODIN-ASC"))) (ARE PROCESSED) FOR (AUTOMATIC DISTRIBUTION ASSIGNMENT))

♦((THE MESSAGE) (IS DISTRIBUTED) TO (EACH ((ASSIGNED)) OFFICE))

♦((THE NUMBER OF (COPIES OF (A MESSAGE) ((DISTRIBUTED) TO (AN OFFICE)))) (IS) (A FUNCTION OF (WHETHER ((THE OFFICE) (IS ASSIGNED) FOR (("ACTION") OR ("INFORMATION"))))))

♦((THE RULES FOR ((EDITING) (MESSAGES))) (ARE) (: ((REPLACE) (ALL LINE-FEEDS) WITH (SPACES)) ((SAVE) (ONLY (ALPHANUMERIC CHARACTERS) AND (SPACES))) ((ELIMINATE) (ALL REDUNDANT SPACES))))

♦(((TO EDIT) (THE TEXT PORTION OF (THE MESSAGE))) (IS) (NECESSARY))

♦(THEN (THE MESSAGE) (IS SEARCHED) FOR (ALL KEYS))

♦(WHEN ((A KEY) (IS LOCATED) IN (A MESSAGE)) ((PERFORM) (THE ACTION ((ASSOCIATED) WITH (THAT TYPE OF (KEY))))))

♦((THE ACTION FOR (TYPE-0 KEYS)) (IS) (: (IF ((NO OFFICE) (HAS BEEN ASSIGNED) TO (THE MESSAGE) FOR ("ACTION")) ((THE "ACTION" OFFICE FROM (THE KEY)) (IS ASSIGNED) TO (THE MESSAGE) FOR ("ACTION"))) (IF ((THERE IS) ALREADY (AN "ACTION" OFFICE FOR (THE MESSAGE))) ((THE "ACTION" OFFICE FROM (THE KEY)) (IS TREATED) AS (AN "INFORMATION" OFFICE))) (((LABEL 0FFS1 (ALL "INFORMATION" OFFICES FROM (THE KEY))) (ARE ASSIGNED) TO (THE MESSAGE)) IF ((REF 0FFS1 THEY) (HAVE (NOT) (ALREADY) BEEN ASSIGNED) FOR (("ACTION") OR ("INFORMATION"))))))

♦((THE ACTION FOR (TYPE-1 KEYS)) (IS) (: (IF ((THE KEY) (IS) (THE FIRST TYPE-1 KEY ((FOUND) IN (THE MESSAGE)))) THEN ((THE KEY) (IS USED) TO ((DETERMINE) (THE "ACTION" OFFICE)))) (OTHERWISE (THE KEY) (IS USED) TO ((DETERMINE) (ONLY "INFORMATION" OFFICES)))))

Figure 1

To illustrate how context is used to complete the partial descriptions in the example, we consider a few cases:

1. *Partial sequencing.* Distribution is never explicitly invoked in the informal specification. However, the first sentence indicates that Assignment is performed to enable the Distribution. Hence, Distribution should be explicitly invoked after Assignment.

2. *Missing operand.* Sentence two indicates that the message should be distributed to certain offices—those that are "assigned." But, as can be determined from other usages in the informal specifications, offices can be "assigned" to either messages or keys. This missing operand can be resolved by remembering that Assignment

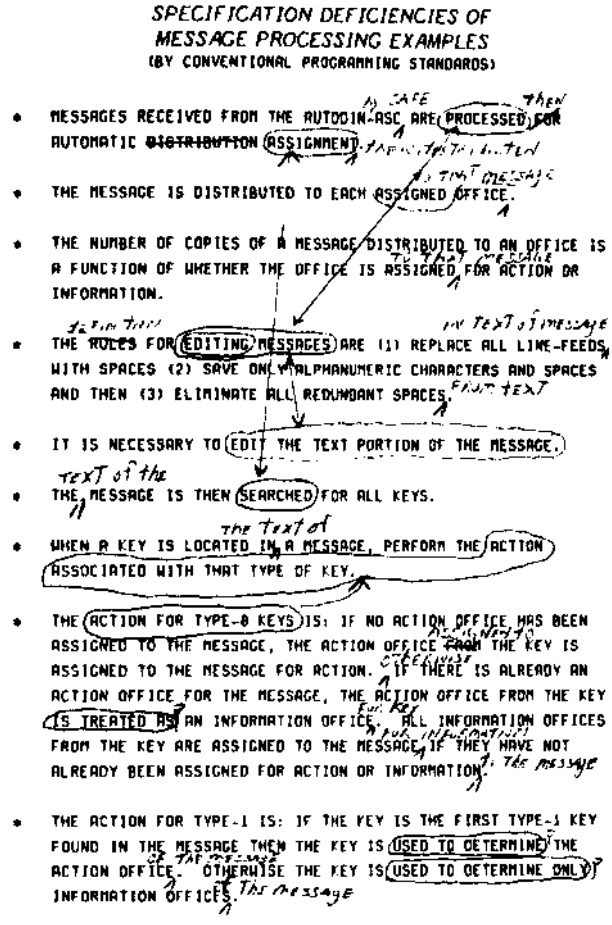


Figure 2

was performed to enable Distribution. Hence, Distribution must use some result of the assignment process. Assignment, from the last two input sentences, assigns offices to the current message. Hence, Distribution must consume offices assigned to that message.

Incomplete reference. Sentence four says to replace all line feeds with spaces. First, replace requires a third operand, some set in which the replacement will occur. Context indicates that this missing operand should be the text of the message parameter of Edit. Second, the use of a plural in the operand of an action which expects a singular operand, indicates an implicit loop. Hence, we have, "for all line feeds, replace the line feed by a space in the text of the message." Now, which line feeds are we concerned with? Only those in the text of the message because they are the only ones which can be replaced. Hence, completing the partial reference, we have "for all line feeds in the text of the message, replace the line feed by a space in the text of the message."

It should be noted that of the approximately 61 decisions which had to be made for this example, all but one were resolved correctly by the prototype system. The message it distributed is the edited one (with all punctuation removed) rather than the original unedited one. The cause of the error is that the system does not understand the difference between an object being changed and its participating in relations with other objects; therefore, it has no concept of the original state of an object and hence does not consider this as a possible completion of any partial reference.

PROGRAM CREATED BY PROTOTYPE SYSTEM

```
(WHENEVER (receive message FROR autodin-asc BY safe)
 DO(odit text OF message)
 (search text OF message FOR (CREATE THE SET OF keys))
 (distribute-processfl message))

(disiribute-process#l (message)
 (FOR ALL (offices assigned TO message FOR ANYTHING)
  (distrtbute-process#2 message office)))

(distribute-process#2 (message office)
 (00 (functional (BOOLEAN (assigned office TO message FOR action))
  (BOOLEAN (assigned office TO message FOR information)))
 TiriES (distribute A copy UH1CH IS A copy OF messago AND located
  AT safe FROM safe TO location OF office)))

(edit (text)
 (FOR ALL line-feeds IN text
  (replace line-feed IN text BY (CREATE SET OF spaces)))
 (keep (union (CREATE THE SET OF alphanumberlc characters IN text)
   (CREATE THE SET OF spaces IN text))
  FROM text)
 (FOR ALL spaces IN text ANO redundant IN text
  (remove space FROM text))

(WHENEVER (locate A key IN text OF message AT POSITION ANYTHING)
 DO (CASE (type OF key)
  (type-8 (type-8-action message key))
   (type-1 (type-1-action message key))))

(type-0-action (message key)
 (IF (NOT (EXISTS action office FOR message))
  THEN (assign THE action officefl FOR key
   TO message FOR action)
 ELSE (treat action office#2 FOR key
   AS information office#2 FOR key
  IN (IF (NOT (assigned office#2 TO message
     FOR action OR information))
   THEN (assign office#2 TO message FOR information))))
 (FOR ALL (offtce#3 assigned TO key FOR information)
  (IF (NOT (assigned office#3 TO message
   FOR action OR information)
   THEN (assign office#3 TO message FOR information))))

(type-1-action (message key)
 (IF key - (key#i UH1CH IS (SEARCH HISTORY FOR FIRST
  (locate type-1 key#I IN text OF message AT position ANY)))
   THEN (determine THE action office FOR message
   BY (type-0-action message key))
  ELSE (determine ONLY THE information office FOR message
   BY (IF (EXISTS action office FOR message)
      THEN (treat action office#I FOR key
         AS information office#I FOR key
         IN (IF (NOT (assigned office #1 TO message
            FOR action OR information))
           THEN (assign office'I TO message FOR information))))
      (FOR ALL office#2 assigned TO key FOR information)
         (IF (NOT (assigned office#2 TO message
           FOR action OR information))
           THEN (assign office#2 TO message
             FOR information))))))
```

Figure 3

This capability can clearly be added to the system, but the important point is that interpretation errors will occur, just as they do when human intermediaries are used to produce the formal specification. It is therefore essential to provide extensive feedback and assumption-testing facilities so that such errors, when made, can be discovered and corrected by the user.

The second example is from a system for scheduling a satellite communication channel by multiplexing it among several users (subscribers). It specifies the component of the system which receives a schedule (SOL) from the controller of the satellite channel and extracts from it the portions of the

next transmission cycle which have been reserved for a particular subscriber and those portions available to any user (RATS). This information is placed in a transmission schedule used by another component to actually utilize the channel during the allowed periods. Figure 4 gives the informal natural language description. Figure 5 indicates some of the imprecisions contained in this example which must be resolved to obtain the system's formalization of the specification as an operational program (Figure 6). In addition to the process description of Figure 4, we have assumed that the formulas referenced and a structural description of the objects of the domain have been separately specified.

The relevant portions of these specifications are that the SOL is an ordered set of subscriber and RATS entries. Each subscriber entry has subscriber identifier and transmission length fields, while a RATS entry has only the latter. The transmission schedule is a set of entries, each of which is composed of an absolute transmission time and a transmission length. One of these entries is the primary entry of the transmission schedule. Finally, formulas 1 and 2 both take an SOL entry as input and produce, respectively, a relative and an absolute transmission time.

Using the same measures of imprecision as in the first example, we find that this example has about half as many imprecisions.

Number of missing operands	"	7
Number of incomplete references	-	12
Number of implicit type conversion	■	3
Number of terminology charges	=	0
Number of refinement or elaboration	-	0
Number of implicit sequencing decisions	=	4

The example is interesting as a test of the generality of the mechanisms which worked on the first example, and because of the new issues it raises. We will examine each of these to illustrate the range of capabilities added to the prototype to enable it to correctly understand this example and produce the operational program of Figure 6.

```
((THE SOL)
 (IS SEARCHED)
 FOR
 (AN ENTRY FOR (THE SUBSCRIBER))))

(IF ((ONE)
    (IS FOUND))
   ((THE SUBSCRIBER'S (RELATIVE TRANSMISSION TIME))
    (IS COMPUTED) ACCORDING-TO ("FORMULA-1"))))

((THE SUBSCRIBER'S (CLOCK TRANSMISSION TIME))
 (IS COMPUTED) ACCORDING-TO ("FORMULA-2")))

[WHEN ((THE (TRANSMISSION TIME))
      (HAS BEEN COMPUTED))
     ((IT)
      (IS INSERTED)
      AS (THE (PRIMARY ENTRY))
      IN (A (TRANSMISSION SCHEDULE])

[FOR (EACH RATS ENTRY)
    (PERFORM)
    (: ((THE RATS'S (RELATIVE TRANSMISSION TIME))
        (IS COMPUTED) ACCORDING-TO ("FORMULA-1"))
       ((THE RATS'S (CLOCK TRANSMISSION TIME))
        (IS COMPUTED) ACCORDING-TO ("FORMULA-2"])

((THE RATS (TRANSMISSION TIMES))
 (ARE ENTERED)
 INTO (THE SCHEDULE)))
```

Figure 4

* THE SOL IS SEARCHED FOR AN ENTRY ~~FOR~~ THE SUBSCRIBER.
* IF (ONE) IS FOUND, THE SUBSCRIBER'S RELATIVE TRANSMISSION TIME IS COMPUTED ~~ACCORDING TO FORMULA-1~~.
* THE SUBSCRIBER'S CLOCK TRANSMISSION TIME IS COMPUTED ACCORDING TO FORMULA-2.
* WHEN THE TRANSMISSION TIME HAS BEEN COMPUTED, IT IS INSERTED AS THE PRIMARY ENTRY IN A TRANSMISSION SCHEDULE.
* FOR EACH RATS ENTRY, THE RATS'S RELATIVE TRANSMISSION TIME IS COMPUTED ACCORDING TO FORMULA-1 AND THE RATS'S CLOCK TRANSMISSION TIME IS COMPUTED ACCORDING TO FORMULA-2.
* ~~THE~~ RATS TRANSMISSION TIMES ARE ENTERED INTO THE SCHEDULE.

Figure 5

```
(build-transmission-schedule (sol subscriber)
   (CREATE transmission-schedule)
   (search sol FOR A subscriber-entry SUCH THAT
    sid OF subscriber EQUALS sid OF subscriber-entry)
   (IF (locate A subscriber-entry SUCH THAT
    sid OF subscriber EQUALS sid
       OF subscriber-entry IN sol)
    THEN
     (MAKE (RESULT-OF (FORMULA-1 subscriber-entry))
      BE THE relative-transmission-time OF subscriber)
     (MAKE (RESULT-OF (FORMULA-2 subscriber-entry))
      BE THE clock-transmission-time OF subscriber))
   (FOR ALL rats WHICH ARE IN sol
       DO (MAKE (RESULT-OF (formula-1 rats))
             BE THE relative-transmission-time OF rats)
          (MAKE (RESULT-OF (formula-2 rats))
             BE THE clock-transmission-time OF rats))
   (FOR ALL clock-transmission-time OF rats
       DO (MAKE clock-transmission-time BE THE
             transmission-time OF (CREATE transmission-entry))
          (ADD transmission-entry TO transmission-schedule)))

(WHENEVER (MAKE time BE THE clock-transmission-time
    OF subscriber)
      DO (MAKE time BE THE transmission-time
            OF (CREATE transmission-entry))
         (ADD transmission-entry TO transmission-schedule)
         (MAKE transmission-entry BE THE primary-entry
            OF transmission-schedule))
```

Figure 6

1. *Scope of conditional.* In natural language communication the end of a conditional is almost never explicit. Instead, context must be used to determine whether subsequent statements are part of the conditional. In sentence three of the example, the input to formula 2 is the SOL entry found in the previous sentence. Thus, sentence three is really part of the conditional statement.

2. *Implicit formation of relations.* In sentence two, the relative transmission time produced by formula 1 is supposed to be associated with the subscriber. Since that association is not established elsewhere, it is implicitly being established here. Hence this passive construct must be treated as an active one.

3. *Implicit creation of outputs.* In a similar fashion, various sentences establish associations with a transmission schedule (the output of this example) but an instance of one is never explicitly created. Such usage indicated that an implicit creation of the output is required.

A. Expectation failure. In addition to process and structural statements, a specification normally contains expectations about the state of the computation at some point which provide context for people to explain why something is being done or some properties of its result. They also provide some redundancy against which an understanding of the specification can be checked. In the example, one of these expectations (that all of the components of the entries of the output have been produced) fails, which indicates either a misunderstanding of the specification or an inconsistency Or incompleteness. In this case, both our example and the actual specification from which it was drawn are incomplete; they fail to describe how the length field of the entries of the transmission schedule are calculated from the inputs.

7. *DESCRIPTION OE THE PROTOTYPE SYSTEM*

The prototype system is structurally quite simple. It has three phases (Linguistic, Planning, and Meta-Evaluation) which are sequentially invoked to process the informal specification. Each phase uses the results of the previous phases, but no capability currently exists to reinvoke an earlier phase if a difficulty is encountered. Hence, either ambiguity must be resolved within a phase or the possibilities passed forward to the next phase for it to resolve.

We will describe the prototype system by working backward from the goal through the phases (in reverse order) toward the input to expose the system design and provide context for understanding the operation of each phase.

The goal of the system is to create a formal operational specification from the informal input, which means that it must complete each of the partial descriptions in the input to produce the output. In general, each partial description has several different possible completions, and a separate decision must be made for each partial description to select the proper completion for it.

Based on the partial description and the context in which it occurs, an a priori ordered set of possible completions is created for each partial description. But one decision cannot be made in isolation from the others; decisions must be consistent with one another and the resulting output specification must make sense as a whole. Since the output is a program in the formal specification language, it must meet all the criteria for program well-formedness. Fortunately, programs are highly constrained objects (one reason they are so hard to write), so there are many well-formedness criteria which must be satisfied.

This provides a classical backtracking situation [12], since there are many interrelated individual decisions that in combination can be either accepted or rejected by some criteria (the well-formedness rules). In such situations, the decisions are made one at a time in some order. After each decision the object (program) formed by the current set of decisions is tested to see if it meets the criteria (well-formedness rules). If it does, then the next decision is made, and so on, until all the decisions have been made and the result accepted. If at any stage the partially formed result is rejected, then the next possibility at the most recent decision point is chosen instead and a new result formed and tested as before. If all possibilities have been tried and rejected for the most recent decision point, then the state of the decision-making process is backed up to that existing at the previous decision point and a new possibility chosen. This process will terminate either by finding an acceptable solution (formal specification) or by determining that none can be found. The resulting object (program) is an acceptable solution (formal specification) f*or* the problem (informal specification).

The order in which decisions (rather than the order of alternatives within a decision) are made should be chosen to maximize early rejection of infeasible combinations of decisions. This requires that the rejection criteria can be applied to partially determined objects. The preferred decision order is clearly dependent on the nature of the acceptance/rejection criteria.

We now let the nature of the well-formedness criteria determine the structure of the prototype system so that the early rejection possibilities inherent in the criteria can be utilized. The criteria fall into three categories: dynamic state-of-computation criteria, global reference criteria, and static flow criteria. Each of these categories must be handled differently.

The dynamic state-of-computation criteria are based only on the current "state" of the program and its data base (e.g., "the constraints of a domain must not be violated" and "it must be possible to execute both branches of a condition"). They require that all decisions that affect the computation to that point (but not beyond) must be made before the criteria can be tested. Thus, if decisions could be made as they are needed by the computation of the program and the program "state" examined at each stage of the computation, then the dynamic state-of-computation criteria could be used to obtain early rejection of infeasible decisions.

This is exactly the strategy adopted in the design of the prototype system. However, since no actual input data is available for the program to be tested, and since the program must be well-formed for a variety inputs, symbolic inputs rather than actual inputs are used. Instead of actual execution, the program is symbolically executed on the inputs, which provides a much stronger test of well-formedness than would execution on any particular set of inputs.

However, completely representing the state of the computation as a program is symbolically executed is very difficult (e.g., determining the state after execution of a loop or a conditional statement) and more detailed than necessary for the well-formedness rules. Therefore, the prototype system uses a weaker form of interpretation, called Meta-Evaluation, which only partially determines the program's state as computation proceeds (e.g., loops are executed only once for

some "generic" element, and the effects of THEN and ELSE clauses *are* marked as POSSIBLE, but are not conditioned by the predicate of the If). This Meta-Evaluation process is much easier to implement and still provides a wealth of run-time context used by the acceptance/rejection criteria to determine program well-formedness.

The global referencing criteria (such as "parameters must be used in the body of a procedure") test the overall use of names within the program and thus cannot be tested until all decisions have been made. They are tested only after the Meta-Evaluation is complete.

The final category of criteria, static flow (e.g., "items must be produced before being consumed" and "outputs must be produced somewhere"), are more complex. The Meta-Evaluation process requires a program on which to operate, which may contain partial descriptions that the Meta-Evaluation process will attempt to complete by backtracking. This program "outline" is created from the informal input for the Meta-Evaluation process by the flow analysis, or Planning, phase, which examines the individual process descriptions and the elaborations, refinements, and modifications of them in the input, then determines which pieces belong together and how the refinements, elaborations, and modifications interact. It performs a producer/consumer analysis of these operations to determine their relative sequencing and where in the sequence any unused and unsequenced operations should occur. This analysis enables the Planning phase to determine the overall operation sequencing for the program outline from the partial sequencing information contained in the input. It uses the data flow well-formedness criteria and the heuristic that each described operation must be invoked somewhere in the resulting program (otherwise, why did the user bother to describe it?) to complete the partial sequence descriptions.

If the criteria are not sufficiently strong to produce a unique program outline, the ambiguity must be resolved either by interacting with the user or by including the alternatives in the program outline for the Meta-Evaluation phase to resolve as part of its decisionmaking process. In the prototype system, the Meta-Evaluation phase is prepared to deal with only minor sequencing alternatives such as the scope of conditional statements (If a statement following a conditional assumes a particular value of the predicate, it must be made part of one of the branches of the conditional.) and demons (Are all situations which match the firing pattern of a demon intended to invoke it or only those which arise in some particular context, and if so what context?). Major sequencing issues—such as whether one statement is a refinement of another or not—that cannot be resolved by the Planning phase must be resolved by the user before the Meta-Evaluation phase.

Both the Planning and Meta-Evaluation phases use a structural description of the application domain to provide context for their program execution, and inference rules which define relation inter-dependencies in the process domain. This structural base is the application-specific foundation upon which the Planning and Meta-Evaluation phases rest, and must be provided before they are invoked. It contains all the application-specific contextual knowledge. It augments the system's built-in knowledge of data flow and program well-formedness and enables the system to be specialized to a particular application and to use this expertise in conjunction with its built-in program formation knowledge to formalize the input specification.

The construction of a suitable application-specific structural base is itself an arduous, error-prone task. Furthermore, our study of actual program specifications indicated that most of the structural information was already informally contained in the program specification. We therefore decided to allow partial descriptions in the specification of the structural base and to permit such descriptions to be intermixed with the program specification.

Since we *are* concerned only with the semantic issues raised by using partial descriptions in the program specification, the system uses a parenthesized version of the natural language specification as its actual input to avoid any syntactic parsing issues. This parenthesized input does not affect the semantic issues we have discussed.

The first tasks, then, of the system are to separate the process descriptions from the structural descriptions, to convert both to internal form, and to complete any partial structural descriptions. These tasks comprise the system's Linguistic phase, which precedes the other two.

If a formal structural base already exists for some application, then, of course, it is loaded first and is augmented by and checked for consistency with any structural statements contained within the program specification.

Thus, in chronological order (rather than the reverse dependence order used above), the system's basic mode of operation consists of reading an input specification, separating it into structural and processing descriptions; completing the structural descriptions and integrating the result into any existing structural base; determining the gross program structure by producer/consumer analysis during the Planning phase; and, finally determining the final program structure through Meta-Evaluation.

REFERENCES

1. Balzer, Robert, Neil Goldman, and David Wile, "On the Transformational Implementation Approach to Programming," *2nd International Conference On Software Engineering,* October 1976, p. 337.

2. Bell, Thomas E. and David Bixler, "An Extendable Approach to Computer-Aided Software Requirements Engineering," *2nd International Conference on Software Engineering,* October 1976, p. 70.

3. Teichroew, Daniel and Ernest Allen Hershey, III, "PSL/PSA A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *2nd International Conference on Software Engineering,* October 1976, p. 2.

4. Jones, Anita K. and Barbara H. Liskov, "A Language for Controlling Access to Shared Data," *SEC Supplement,* 1976, p. 68.

5. Wulf, Wm. A. and Mary Shaw, "An Introduction to the Construction and Verification of Alphard Programs," *2nd International Conference On Software Engineering,* October 1976, p. 390.

6. Lampson, B. W. and J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek, "Report on the Programming Language Euclid," Xerox Research Center, Palo Alto, August 1976.

7. Snowdon, R. A. , *PEARL: An interactive system for the preparation and validation of structured programs.* Computing Laboratory, University of Newcastle Upon Tyne, November 1971.

8. Schwartz, J. T., *On Programming, An Interim Report on the SETL Project,* Computer Science Department, Courant Inst. Math. Sci., New York University, 1973.

9. Guttag, John V., Ellis Horowitz, and David R. Musser, "The Design of Data Type Specifications," *2nd International Conference On Software Engineering,* October 1976, p. 414

10. CODASYL, Data Base Task Group, April 1971 Report, ACM, New York.

11. Elschlager, R., "Overview of a Natural Language Programming System," Unpublished Report, CS Department, Stanford University, February, 1977.

12. Gerhart, Susan L. and Lawrence Yelowitz, "Control Structure Abstractions of the Backtracking Programming Technique," *2nd International Conference On Softu/ar* Engineering,* October 1976, p. 391.